

Look It Up!

Over the years, I've written a number of articles about how to efficiently compute functions of one sort or another. We've done trig functions, logarithms, exponentials, and a few other assorted functions. In each situation, I presented an algorithm that would give you the function result with a minimum of math operations.

Sometimes an algorithm, no matter how efficiently implemented, is still not the best way to go. In very demanding real-time situations, even the multiplication and division needed to generate, say, a sine function is too much for the computing power and time available. In such a case, the only other possible solution is a table lookup. Sometimes, it's easier and faster to simply look up stored values rather than computing them.

Anyone who ever took high school trigonometry should have no trouble understanding the concept of table lookups. Remember all those horrible columns of numbers in the back of your trig text? Remember how much fun you had looking up the sine of 27.328°? Or worse yet, the arcsine? That's the kind of thing we're talking about here. Instead of having a magic formula that computes the sine for any given angle, we can simply store that same table into memory and look up the needed value. Yes, this approach may need lots of storage space. The tradeoff we generally arrive at is speed vs. memory size. A table lookup, done properly, can be much faster than the evaluation of a polynomial, but will require a lot more memory.

In this column, we'll explore the concept of table lookups, including the tradeoff issues involved. By the time we're finished, you should have two things: one more set of tools in your toolbox in the form of table lookup algorithms, and enough insight into the

In this month's column, we'll explore the concept of table lookups, including the tradeoff issues involved.

problems to help you decide which approach is more appropriate for your application. We'll begin with the most dumb algorithm I can think of and go from there to more effective and efficient algorithms.

QUICK, DIRTY, AND DUMB

Let's think about that theme of trig functions a bit more. The size of the table will obviously depend on the resolution we need in the angle. Table 1 shows the sine function of angles from zero through 90°. For the coarse resolution I've shown, with 1° increments, the table is certainly small enough, needing only 90 entries. If we only need values of the sine to this degree of accuracy, we can write a quick-and-dirty algorithm that gives the results in a micro-jiffy. Here's the code:

```
float sine_table[91] = { ...};
float sine(float x){
    return table[(int)x];
}
```

Yes, this is really dirty code. For one thing, the angle is in degrees rather than the usual radians. For another, we have absolutely no protection against violation of the array bounds. Finally,

there is absolutely no interpolation between points. Still, as a starting point, the code is not really all that bad a solution. A test with $x = 10.3276^\circ$ gives a return value of 0.173648 (the sine of 10° , as expected). For the record, that's an error of only 0.56%, which is horrendous for serious math crunching but probably more than adequate for, say, computer game graphics. So, if you must have a sine in a hurry and don't care much for accuracy, look no further.

In reality, of course, we know that the angle argument of the sine function can exceed the bounds 0..90°. But we also know about certain trig identities like:

$$\begin{aligned}\sin(90-x) &= \cos(x) \\ \sin(180-x) &= -\sin(x)\end{aligned}\quad (1)$$

so we can add a wrapper (just as we'd do if we were using a polynomial) to ensure that the bounds will not be violated. Still, a proper function should protect against such things. The usual rule is to return the limiting values in the table, if the bounds are exceeded:

```
float sine(float x){
    int i = (int)x;
    if(i < 0) return table[0];
    if(i > 90) return table[90];
    return table[(int)x];
}
```

We can make this code smaller and faster by encapsulating the if statements into one statement:

```
float sine(float x){
    int i = max(0, min(90, (int)x));
    return table[i];
}
```

This approach is more efficient, because the min and max functions are typically implemented as macros.

TABLE 1
A table of sines.

x	sin x
0	0
1	0.0174524
2	0.0348995
3	0.052336
4	0.0697565
5	0.0871557
6	0.1045285
7	0.1218693
8	0.1391731
9	0.1564345
10	0.1736482
11	0.190809
12	0.2079117
13	0.2249511
14	0.2419219
15	0.258819
16	0.2756374
17	0.2923717
18	0.309017
19	0.3255682
20	0.3420201
21	0.3583679
22	0.3746066
23	0.3907311
24	0.4067366
25	0.4226183
26	0.4383711
27	0.4539905
28	0.4694716
29	0.4848096
30	0.5
31	0.5150381
32	0.5299193
33	0.544639
34	0.5591929
35	0.5735764
36	0.5877853
37	0.601815
38	0.6156615
39	0.6293204
40	0.6427876
41	0.656059
42	0.6691306
43	0.6819984
44	0.6946584
45	0.7071068
46	0.7193398
47	0.7313537
48	0.7431448
49	0.7547096
50	0.7660444

(continued)

x	sin x
51	0.777146
52	0.7880108
53	0.7986355
54	0.809017
55	0.819152
56	0.8290376
57	0.8386706
58	0.8480481
59	0.8571673
60	0.8660254
61	0.8746197
62	0.8829476
63	0.8910065
64	0.898794
65	0.9063078
66	0.9135455
67	0.9205049
68	0.9271839
69	0.9335804
70	0.9396926
71	0.9455186
72	0.9510565
73	0.9563048
74	0.9612617
75	0.9659258
76	0.9702957
77	0.9743701
78	0.9781476
79	0.9816272
80	0.9848078
81	0.9876883
82	0.9902681
83	0.9925462
84	0.9945219
85	0.9961947
86	0.9975641
87	0.9986295
88	0.9993908
89	0.9998477
90	1

GENERALIZING

Of course, we would like to generate functions other than sine functions, and most others won't be so accommodating as to have input values that align with the table indices—that is, 0, 1, 2...90. However, both objections are easy to overcome. First, we'll define a general table size:

```
#define MAX_INDEX 90
```

and a more general table:

```
double table[MAX_INDEX+1];
```

Next, we'll define a scale factor that relates the function argument x to the table index i . We can do this by specifying a delta, the step in x between table entries, or the range of the entire table. The delta works better, but we can define it indirectly. For this purpose, C++ `const` values work better than `#defines`:

```
const double range = pi/2;
const double delta = range/MAX_INDEX;
```

Now our lookup routine is beginning to look more like a general-purpose function:

```
float lookup(float x){
    int i = max(0, min(MAX_INDEX,
        (int)(x/delta + 0.5)));
    return table[i];
}
```

The 0.5 addend rounds the value computed from x to the nearest integer rather than simply truncating it. With this change, the maximum error in our simple implementation of the sine function is under 1% through the entire range. Not bad for a "stupid sine trick." Of course, the mileage for your own function may vary.

MORE ACCURACY?

But what if that 1% is still not enough? In that case, we can either make the table larger, with more entries and a smaller increment between them, or we can go to a fancier algorithm. We'll end up doing both. How many entries will we need? Well, the tables in those dear old trig books usually gave the angles with deltas of either 0.01° (needing 9,000 entries) or one minute of arc (needing 5,400 entries). Tables of the trig functions to thousandths of a degree, which are called five-place tables, are fairly common, and I've even seen one book of six-place tables. (It was a very large book!) The engineer who owned it treated it as though it were pure gold.

Using a table of sines with 0.01° increments, our simple algorithm can give us a worst-case error (at zero) less than 0.01%. That six-place table has a worst-case error of only 0.00008%. So you see, even our quick-and-dirty algorithm will perform if given a large enough table. It really is a size vs. speed tradeoff, or rather a size/speed/accuracy tradeoff. Use a large enough table, and you can get virtually any accuracy you need. The speed of the algorithm is the same whether you have a table of 10 entries or 10,000. That characteristic is the big advantage of table lookups.

While our super-quick algorithm will not give very high accuracy, it's often good enough for certain tasks. For those of you interested in using it, Figure 1 shows a graph of the worst-case error plotted against table size. Just pick the error you can live with, and go for it.

Eventually, memory size needed for this simple algorithm gets out of hand. A six-place table needs 900,000 entries, which is serious memory space. So what do we do when we need more accuracy but can't spare the memory? I'm sure you already know the answer because, again, you learned it in trig class: We interpolate. We must take the tried-and-true approach of going to a higher-order algorithm. In this case, we'll use a first-order approximation, or linear interpolation, instead of the "nearest entry" algorithm that could be thought of as "zeroth" order.

INTERPOLATION

The concept of linear interpolation is quite simple and straightforward. Consider the two data points of Figure 2. Instead of merely taking one or the other value, we'll fit a straight line between the two points and compute the function as though it lay along that line. In general, it will not; it will be a curve such as the one shown. As you can see, linear interpolation will always be a much closer approximation than our "nearest entry" solution.

We can think of linear interpolation in a few different ways. The rigorous

FIGURE 1
Error in the simple lookup.

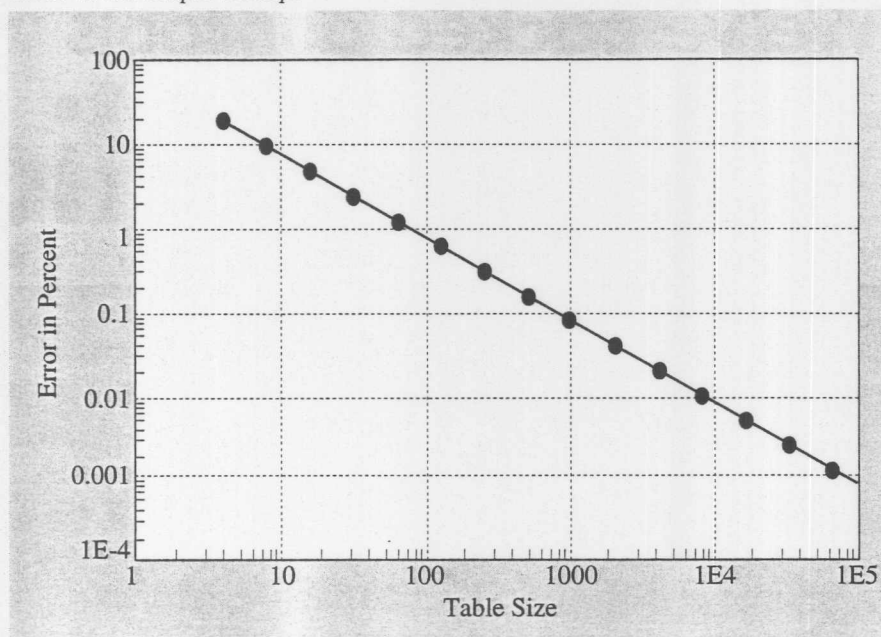
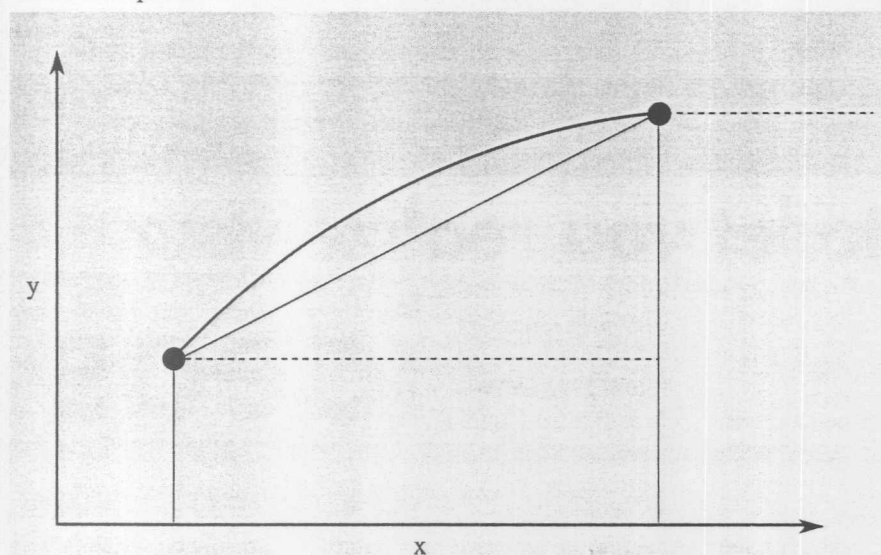


FIGURE 2
Linear interpolation.



explanation comes from the Taylor series expansion:

$$y(x) = y(x_0) + \left[\frac{dy}{dx} \right]_0 (x - x_0) + \frac{1}{2!} \left[\frac{d^2y}{dx^2} \right]_0 (x - x_0)^2 + \frac{1}{3!} \left[\frac{d^3y}{dx^3} \right]_0 (x - x_0)^3 + \dots \quad (2)$$

All those zero subscripts imply that the values of x and y and all the derivatives must be evaluated at the point x_0 . Truncating this series to the first order gives the linear approximation:

$$y(x) \approx y(x_0) + \left[\frac{dy}{dx} \right]_0 (x - x_0) \quad (3)$$

In the absence of any knowledge of derivatives, we can approximate the

derivative by divided differences:

$$\left[\frac{dy}{dx} \right]_0 \approx \frac{y_1 - y_0}{x_1 - x_0} \quad (4)$$

Our interpolation formula, in useful form, then becomes:

$$y(x) \approx y_0 + \left(\frac{y_1 - y_0}{x_1 - x_0} \right) (x - x_0) \quad (5)$$

Of course, there is nothing magic about the subscripts 0 and 1. Any two successive points will do, and in practice we use whichever two points are closest to the value of x we're working with.

That's the mathematically rigorous explanation of interpolation. Feel free to use it to impress your coworkers. The simpler, equally valid way to look at it is that, since the "curve" connected the points in Figure 2 is a straight line, the change in y over that straight line will be proportional to the change in x . Writing a proportionality relationship, we have:

$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0} \quad (6)$$

which leads us right back to Equation 5, but without the intimidating, if impressive, math.

Using Table 1 with 1° steps, this algorithm gives a value of 0.179270078 for the sine of 10.3276 degrees. My calculator says that the actual value is 0.179276142, implying an error of only 0.0006%. That's not bad at all for such a simple algorithm, and 1,000 times better than 0.56%, our same function without interpolation.

Because the sine function happens to be nearly linear for small angles, we'd expect the error to be small for an angle around 10° . For this function, the worst-case error happens to occur near 90° , and is given by:

$$\epsilon_{\max} \approx \frac{\delta^3}{24} \quad (7)$$

For your convenience, I've also plotted

this error as a function of table size in Figure 3. Qualitatively, it looks identical to Figure 1, but notice the vastly different scale for the errors.

LOOKUP VS. FUNCTION GENERATION

In the past, I've given you algorithms for the sine function involving power series that included at the least third- or fifth-order terms. So how are we getting away here with only a first-order approximation? Once more, we are willing to pay the price in storage for all those table entries. The polynomial formulas must give accurate values over a range of at least 22.5° and sometimes more. Here, we only need accuracy between tabular points—a much smaller range. Again, we're burning memory storage and perhaps compromising our accuracy to get a very fast function. A story might help illustrate the point.

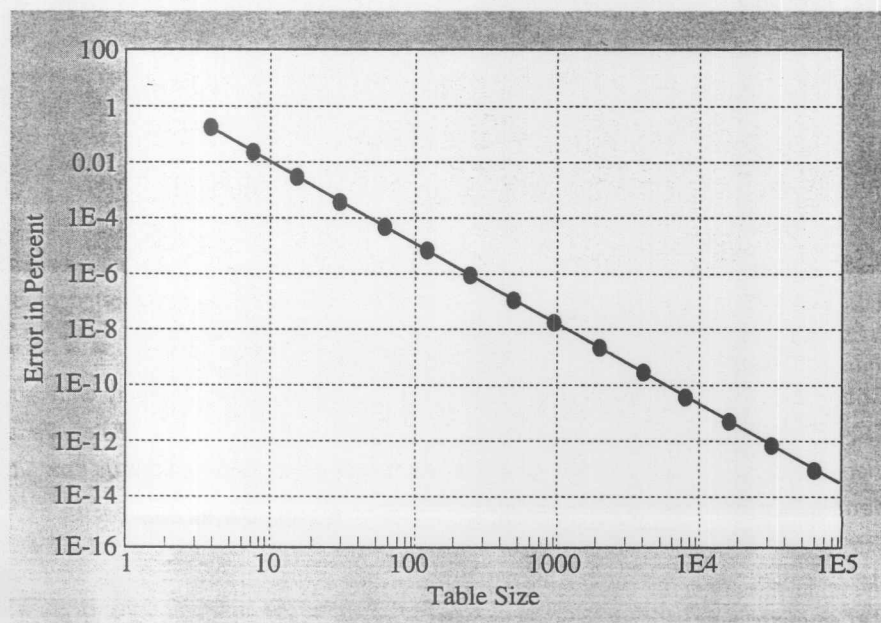
Years ago, I worked with folks who used dynamic simulations for spacecraft launch vehicles like the Saturn V. In those days, they all carried their programs around in boxes of punched cards, and a good percentage of the size of those programs was made up by the tables for atmospheric data and other tables. A few years after that, I discovered that the U.S. Standard

Atmosphere we were using was really defined as a much shorter table, with perhaps 10 entries in it. These are the temperatures at certain altitudes and, by definition of the model, the temperature between points is either constant or linear with altitude.

For such a model, it turns out that we can compute the air density and other properties analytically, so there's no need for interpolation. Or, the interpolation is a model-based one that exactly matches the standard between tabular points. I wrote a subroutine to calculate these properties as a function of altitude. Instead of a box or two of tabular data, my subroutine had the data built in as a declaration, and the whole subroutine was only about 10 cards of object code. I was pretty proud of myself, and I must admit to looking with some disdain at those engineers who were still carrying around all those boxes of cards.

As it turns out, I had completely missed the point. Those engineers were no dummies. They knew how the U.S. Standard Atmosphere was defined, and they could implement an analytically correct interpolation. But they had realized something else: They couldn't afford the computer time. The analytical solution I'd used between points

FIGURE 3
Error with interpolation.



involved a difference of exponential functions, and they figured out long before I did that the two multiplications associated with interpolation are a lot faster than the two function calls to a library function that I was using. They had not just blindly used what was given to them but had made a conscious decision to use a huge table to gain computing speed.

MECHANIZING THE LOOKUP

The code to provide interpolation is a straightforward translation of Equation 5. We must only make sure that neither of the indices involved (we have two now) will ever exceed the array bounds. In other words, we'd like the two tabular points used to straddle the x argument. The new function is shown in Listing 1. With only four executable statements, it's still pretty short.

Note the subtle differences in the code: We're now limiting the range of the index to `MAXINDEX-1`, just to be sure we can use the `table[i+1]` term later. I've also removed the rounding factor, since we want to always keep the argument between the tabular values. By the way, I've overloaded the function `mod()` in my library. If you aren't using it, you should use the C standard library routine `fmod()`.

The lookup function as given in Listing 1 is complete and should work well enough if you're only looking up one function. However, it will only work if you make the table either global or static within function `lookup()`. This is a reasonable solution if you're sure you're never going to be looking up things in more than one table. A more useful and more toolbox-oriented solution would be to design the lookup function to let the table be passed in as a parameter.

One way to do this, the C choice, would be to pass a pointer to the table. Or, using C++, we could pass a reference variable. However, don't forget that other data must be available to the lookup routine, notably the size, range, and delta variables. A much classier and certainly more bulletproof approach might be to make the whole

LISTING 1

Lookup with interpolation.

```
double lookup(double x){
    int i = max(0, min(MAXINDEX - 1,
        (int)(x/delta)));
    double dx = mod(x, delta);
    double slope = (table[i+1] -
        table[i])/delta;
    return table[i] + slope * dx;
}
```

table, including its related variables, a structure. Even classier (literally), why not encapsulate the whole thing, including the lookup routine, in a C++ object? In doing so, we can guarantee that nobody messes with those all-important range and limit variables. Here's my take on what such an object should look like:

```
class Table{
    int N;
    double *p;
    double Range;
    double delta;
public:
    Table(int maxindex, double range);
    ~Table();
    // I/O routines here
    double lookup(double x);
};
```

The complete class, including the code for each function, is shown in Listing 2. As you can see, the table has the same general structure as a vector, and we could have used our class `Vector` as a parent object. However, we don't really want to (or allow someone else to) do things like dot and cross products between two tables. The class `Vector` comes complete with so much extraneous garbage in the form of math operations that can be performed on it, we'd end up having to jump through hoops to make these functions inaccessible for this new class. It's better just to make the table a separate class, or at least allow the two classes to inherit from a common parent.

In the real world, we need somehow to get data values into the table. For the purposes of this column, I included in the class a simple little function `init()`

LISTING 2

The object-oriented approach.

```
class Table{
    int N;
    double *p;
    double Range;
    double delta;
public:
    Table(int maxindex, double range);
    ~Table();
    void init();
    void dump();
    double lookup(double x);
};

Table::Table(int maxindex, double range){
    N = maxindex + 1;
    p = new double[N];
    assert(p != 0);
    Range = range;
    delta = range/maxindex;
}

Table::~Table(){
    delete[] p;
}

void Table::init(){
    for(int i=0; i < N; i++){
        p[i] = sin(to_radians((double)i));
    }
}

void Table::dump(){
    for(int i=0; i < N; i++){
        cout << i << " " << p[i] << endl;
    }
}

double Table::lookup(double x){
    int i = max(0, min(N-2,
        (int)(x/delta)));
    double dx = mod(x, delta);
    double slope = (p[i+1] -
        p[i])/delta;
    return p[i] + slope * dx;
}
```

that would fill the table with values from the sine function. I also wrote a function that would dump the table for testing. But in real applications, such tables will typically come from disk files. This looks like a fine opportunity for overloaded stream I/O operators.

EVEN MORE EFFICIENCY

Throughout this column, I've made two tacit assumptions. The first is that the data we're using is a table approximating some analytic function like the sine. The second assumption is that every table has data points tabulated at nice, evenly spaced points with a step size given by delta. But many real applications come from data collected by experiment in the lab or by measurements and definitions such as the U.S. Standard Atmosphere. This kind of data rarely comes to us equally spaced, but rather as pairs of x- and y-coordinates spaced however nature dictates. A table lookup on that kind of data requires a totally different approach that we'll address next month.

Before I close, I'd like to go back to that notion of using a table lookup for the sine function and give you some parting thoughts for getting even more accuracy and efficiency out of it.

Many real applications come from data collected by experiment in the lab or by definitions.

Take another look at Equation 5. The term in parentheses:

$$\left(\frac{y_1 - y_0}{x_1 - x_0} \right)$$

representing the slope of the curve contains only the values of x and y at

the tabular points. So this term is a constant for every single interval in the table. That being the case, it's dumb to recompute it each time. Instead, let's just store it along with the tabular data. We've already agreed to use memory space to save time, and here's another opportunity to do so. The revised class looks like this:

```
class Table{
    int N;
    double *p_data, *p_slope;
    double Range;
    double delta;
public:
    ...
    double lookup(double x);
};
```

And the revised table lookup function looks like this:

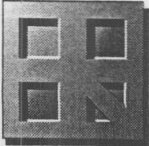
```
double Table::lookup(double x){
    int i = max(0, min(N-2, (int)
```



Unleash the Power of **PromICE**

- ❖ Connect via ROM Socket; DIP; PLCC; SMT
- ❖ Emulate ROMs up to 16MBit in Size
- ❖ Fastest Downloads Available:
Parallel; Serial; Ethernet
- ❖ Run Industry Standard Debuggers
- ❖ Target Processor Independent
- ❖ Support 3Volt Targets
- ❖ Host Software Sources Included
- ❖ Shielded Cables for Reliable Operation
- ❖ 30-day Money-Back Guarantee;
1 Year Warranty!
- ❖ Unlimited Phone Support; 24hr BBS

**Call Today 1-800-PROMICE
(1-800-776-6423)**



Grammar Engine Inc.
921 Eastwind Dr., Suite 122 • Westerville, OH 43081
614/899-7878 • Fax 614/899-7888

CIRCLE # 13 ON READER SERVICE CARD

**Design in:
TCP/IP,
along with
SNMP & PPP
Protocols
to make your
Embedded
Applications
Fly.**



**Highly
Portable
INTERNET
Protocols
added to your
System Design.**



Call Today For More Information:
1-800-541-9508
805-484-2128 • FAX 805-484-3929



**PORTING SERVICES
FULL PORTING GUIDE
DOCUMENTATION
PROTOCOL TRAINING
CONSULTATION
ROYALTY-FREE OPTIONS**

CIRCLE # 46 ON READER SERVICE CARD


```

(x/delta));
double dx = mod(x, delta);
return p_data[i] + p_slope[i] * dx;
}

```

You can't get simpler or more efficient than that! As long as we're on the subject, I'll mention an application where we used this technique and even went it one better.

We needed a cosine function very quickly, and also one that was very quick, for an embedded application. The processor didn't support floating point, and we wouldn't have been able to afford the time even if it did. We implemented it as a table lookup just as you've seen it here, but with a twist. As we said at the beginning of this piece, you don't need to store the value of the sine function for all 360° of the circle. You can use identities like Equation 1 to reduce the range of the function to only that part where the values are unique, that is, zero to 90° as shown in

Table 1. Functions that compute the sine analytically use similar identities to reduce the range even further to 45°, 22.5°, or even 15°. But all these reductions require range-checking and modifying of the results, and that spells time. To avoid that, we simply stored the cosine values for the entire circle. In keeping with the philosophy inherent in table lookups, we willingly used four times the memory space to save testing the argument and adjusting the result.

In this application done in integer arithmetic, we also used another trick: We expressed the angle in "pirads" (pi radians) rather than degrees or radians. In two's complement integer arithmetic, a 16-bit integer wraps around from 0xffff back to zero as it passes through its full range. That's precisely the behavior we'd like to see in an angle. When we express an angle in pirads, it behaves just the way we'd like with no extra wrappers to deal

TABLE 2
Angles in pirads.

Angle, degrees	Angle, pirads (in hex)
0	0x0000
45	0x2000
90	0x4000
135	0x6000
180	0x8000
225 = -135	0xa000
270 = -90	0xc000
315 = -45	0xe000
360 = 0	0x0000

with the range. The behavior of an angle expressed in pirads is shown in Table 2. (The sign behavior, in which an angle greater than 180° can be considered either positive or negative, is also preserved.)

Expressing the angle in pirads gave us one more important benefit: It reduced the computation of the index, *i*, and the offset, *dx*, to trivialities.

QUICK!

THINK OF A TOOL THAT GETS YOUR REAL-TIME PRODUCT TO MARKET FASTER

Having trouble? So were we. That's why we, as real-time developers, created the ObjecTime™ toolset to reduce time-to-market by automating key development activities. With ObjecTime™, to rapidly develop complex event-driven software you can:

- Create reusable components and component frameworks
- Execute and validate graphical requirements and design models throughout the development cycle
- Automatically generate complete production-quality code from design models. These are not just code "headers" or skeletons, but include the hardest parts to get right—complex component behavior and component interconnection and interaction. Target operating systems include VRTX, pSOS+, VxWorks, HP-RT, HP-UX, AIX, and Solaris.

While other companies are debating the merits of academic methods and traditional CASE diagramming tools, your real-time product is shipping!



ObjecTime supports the ROOM method authored by Bran Selic, Garth Gullekson, and Paul Ward (co-developer of the Ward-Mellor method)



OBJEC TIME™ ObjecTime Limited
1-800-567-TIME

Available on Sun, HP, and IBM RS/6000 workstations

CIRCLE # 15 ON READER SERVICE CARD

TEAM PARADIGM

We've beefed up our well-respected arsenal with new releases featuring more capacity and powerful new capabilities. Experience the raw power and searing speed of these field-tested development tools yourself. Be ready to kick a little butt and rest assured that Team Paradigm is here to back you up.



NEW!
DEBUG 4.0

I'VE GOT THE PERFECT DEBUG COMPLEMENT. YEAH, MR. 'SEARCH & DESTROY' IS A REAL BUG KILLER, BUT I NEED MAXIMUM AGILITY AND FINESSE. NOTHING MEASURES UP IN HANDLING THE MOST DIFFICULT C/C++ EMBEDDED APPLICATIONS.

HEY DUDES!
CHECK OUT THE NEW PARADIGM DEBUG 4.0. IT'S A KILLER APP WITH THE RIGHT BALANCE FOR FINDING AND EXTERMINATING PESKY BUGS. NOTHIN' LIKE RAW FIREPOWER FOR GETTIN' THE JOB DONE -- THE FIRST TIME.

NEW!
LOCATE 5.0

No way this dinky ad'll let us lay all the awesome details on you - especially about DEBUG. So ring us today to get the whole scoop. Call! Or else Doc'll beat ya like a bongo.

'Nuff said.

PARADIGM

Proven Solutions for Embedded C/C++ Developers

1-800-537-5043

Paradigm Systems
3301 Country Club Road, Suite 2214
Endwell, NY 13760

(607) 748-5966
FAX: (607) 748-5968
Internet: 73047.3031@compuserve.com

TO BE
CONTINUED...

© 1994 Paradigm Systems, Inc. All rights reserved.

Because we naturally chose the number of entries to be a power of two, we could simply mask off the angle to get both values. The following code fragment tells the whole story, based on a table with 256 entries:

```
i = theta >> 8;  
dx = theta & 0xff;  
cos = table[i] + (((long) slope[i] * dx)  
    >> 7);
```

The shifting and typecasting was required because we stored the slope with maximum precision, so it would have overflowed under ordinary integer arithmetic. Other than that bit of C-ness required for multiplying scaled integer values, the code is a model of simplicity and speed. Try it.

I should mention one last tweak before closing. If you take another look at Figure 2, you'll see that the actual curve of the function always lies above our straight-line approximation. So even though the linear interpolation closely follows the curve, its errors are not random or balanced, but systematic. For a curve that is convex like the sine function, an ellipse, or a parabola, linear interpolation always gives a result that's closer to zero than the actual curve.

It is possible to avoid systematic errors such as these and, in the process, cut our maximum error in half by tweaking the tabular values so the line segments criss-cross the function curve rather than just touching it. We did that in the real-time cosine function I've just described. Computing the optimal values is a bit of a pain, but it's a one-time pain, and it's worth the effort. We'll be talking about things like optimizing coefficients in future columns. Until then. **ESP**

Jack W. Crenshaw is a staff scientist at Invivo Research in Orlando, FL. He did much early work in the space program and has developed numerous analysis and real-time programs. He holds a PhD in physics from Auburn University. Crenshaw enjoys contact and can be reached via e-mail at 72325.1327@compuserve.com.

More on Table Lookups

Last month, we talked about algorithms for looking up data in tables. We started with a very simple scheme of taking the nearest tabular value, and we quickly progressed to linear interpolation. We learned that looking up data in a table can be much faster than computing a formula for a given function. Also, we can get all the accuracy we need using a table lookup if we're willing to store a large enough table. The tradeoffs are familiar: size vs. speed vs. accuracy.

All the work we did last month was based upon the assumption that we could generate a table on a grid of evenly spaced points. But not all problems are so accommodating. Sometimes, we're given the data—perhaps experimental data—in the form of tables that are not evenly spaced in the independent variable. This situation drastically complicates and slows down the algorithm. This month, we'll consider that case. Before diving into this subject, though, I'd like to continue awhile with the example I closed with last month and take a look at computing trig functions via table lookup.

A UNIFYING PRINCIPLE

Most implementations of trig functions involve evaluating power series with pre- and post-processing to limit the range of the input angle to small values. For the record, the power series for the sine function is:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \dots \quad (1)$$

A straightforward implementation would involve direct evaluation of this function. Hopefully, we'd be smart enough not to just code this equation verbatim, since that would involve evaluation of all those higher powers

Most implementations of trig functions involve evaluating power series with pre- and post-processing.

of x . Any decent implementation would use Horner's method, and write:

$$\sin x = x \left(1 - \frac{x^2}{2 \cdot 3} \left(1 - \frac{x^2}{4 \cdot 5} \left(1 - \frac{x^2}{6 \cdot 7} (1 - \dots) \right) \right) \right) \quad (2)$$

As with most power series, this one is very accurate when x is small, but the accuracy goes down the tubes as x increases. In practice, we must play tricks to limit the range of x . We do this using identities like the double-angle formulas that let us get good accuracy with only three to five terms in the series. However, these tricks cost computer time. A typical sine function implemented in your favorite compiler is likely to have three or four tests to limit the angle and corresponding code to adjust the final result.

Last month, we took a completely different approach based on the idea of using a table lookup. The slope of the curve that we use in the linear interpolation is given by:

$$m_n = \frac{f_{n+1} - f_n}{x_{n+1} - x_n} \quad (3)$$

Since all the values in this formula are constant tabular values, we don't need to compute the slope at run time. Instead, we can compute it in advance and store it in the table along with f_n . The result is equivalent to evaluating a linear polynomial with tables for the coefficients:

$$\sin x = a_n + b_n(x - x_n) \quad (4)$$

Recall that n represents the number of steps past the beginning of the table. We can calculate it by dividing the distance from the beginning of the table $x - x_0$ by the grid spacing h . The remainder of this same division represents a fraction of one step, which is the argument of our linear interpolation. The process is especially easy if we meet the following two conditions:

- We express the angle in pirads (pi radians), since the binary representation of the angle covers its full range as the angle goes from zero through 360° .
- The number of table entries is a power of two.

Given these conditions, we can accomplish the division using a shift and mask. For example, if we have 2^8 or 256 table entries, the algorithm is:

```
n = x >> 8;
dx = x & 0xff;
```

At first glance, you might not see much similarity between Equations 2 and 4. The similarity becomes more obvious with a bit more math. The double-angle identity tells us that:

$$\sin(a+b) = \sin a \cos b + \cos a \sin b \quad (5)$$

Let $a = x_n$ and $b = \Delta x$. Then we get:

$$\sin(x_n + \Delta x) = \sin x_n \cos \Delta x + \cos x_n \sin \Delta x \quad (6)$$

A first order approximation, good when Δx is small, gives:

$$\begin{aligned}\cos \Delta x &= 1 \\ \sin \Delta x &= \Delta x\end{aligned}\quad (7)$$

With these values, Equation 6 becomes:

$$\sin(x_n + \Delta x) = \sin x_n + \cos x_n \Delta x \quad (8)$$

This equation has the same form as Equation 4. Comparing them, we not only get reassurance that we're doing the right thing, we get the values for the coefficients:

$$a_n = \sin x_n; \quad b_n = \cos x_n \quad (9)$$

We can use these values to build our table.

We're not just limited to the first-order approximations of Equation 7. We can expand $\sin \Delta x$ and $\cos \Delta x$ to any order we choose, substitute the resulting series into Equation 6, and collect terms. The math may be a little tedious, but the concepts are straightforward and, after all, we only need to do the math once. In general, we'll get a higher-order polynomial in Δx :

$$\sin x = a_n + b_n \Delta x + c_n \Delta x^2 + d_n \Delta x^3 + \dots \quad (10)$$

Again, the best implementation uses Horner's form:

$$\sin x = a_n + \Delta x(b_n + \Delta x(c_n + \Delta x(d_n + \dots))) \quad (11)$$

You might be wondering why the formula contains all powers of x while Equation 1 contains only odd powers. That's because we're not expanding in powers of x but in Δx . When we expand all terms of the form $(x + \Delta x)^n$, the cross-product terms give us all the powers between zero and n .

When we first began looking at table lookups, the situation might have seemed to be a case of either/or: We use either a power series or a table lookup. Now, you can see that it's more of a continuum. At the one extreme, we can use the classical

power series with no table at all, or a degenerate table with only one entry for the coefficients. With this approach, we can expect to do some pre- and post-conditioning to make sure the argument is kept in a small range. At the other extreme, we can do a pure table lookup, with at most a linear interpolation.

In between is a whole continuum of hybrid implementations combining the best features of each extreme. We can use higher-order polynomials, but with fewer terms than we might otherwise need. And the use of a table lets us avoid the end-game logic needed to keep the angle bounded. Instead of setting flags or applying double-angle formulas to adjust the final result, we simply pull the right coefficients out of a table, which is a very efficient process. The next time I have to implement a trig function, I plan to use this approach.

MY FAVORITE FORMULA

If you decide to follow suit, recognize that the tedious expansion of the trig functions I have referred to is unnecessary and not even quite correct. When we did our linear interpolation, we did not compute b_n from:

$$b_n = \cos x_n \quad (12)$$

We didn't even call it b_n . We just computed the slope:

$$m = \frac{f_{n+1} - f_n}{h} \quad (13)$$

which involves only the difference Δf_n . In fact, if you compute the two values, you'll find them to be slightly different. The one obtained from Equation 13 is the correct value because it guarantees that the result yields f_{n+1} when $\Delta x = h$.

We can obtain the higher-order formulas using the same approach without worrying about the math describing the function. To do so, I get to use my favorite formula that connects derivatives, differences, and the z -operator. In the past, I've called it the Rosetta stone of numerical analysis:

$$z = 1 + \Delta = \frac{1}{1 - \nabla} = e^{hD} \quad (14)$$

Here, z is the digital delay operator, Δ and ∇ are the forward and backward difference operators, and D is the derivative operator. Equation 14 provides the connecting link between them all.

Let's see how they apply to our problem. To avoid confusion with the difference operator, let's write our earlier parameter Δx as a fraction of the step size h :

$$\Delta x = \alpha h \quad (15)$$

or:

$$\alpha = \frac{\Delta x}{h} \quad (16)$$

This value of α will be the remainder from the division that produces n .

Taylor's series tells us that:

$$\begin{aligned}f(x_n + \alpha h) &= f(x_n) + \alpha h f'(x_n) + \\ &\quad \frac{\alpha^2 h^2}{2!} f''(x_n) + \dots\end{aligned}$$

or using operator notation:

$$f(x_n + \alpha h) = e^{\alpha h D} f_n$$

Using Equation 14, we can express this in terms of backwards differences:

$$f(x_n + \alpha h) = (1 - \nabla)^{-\alpha} f_n$$

Next, we can expand the term in parentheses in a power series, to get:

$$\begin{aligned}f(x_n + \alpha h) &= \left[1 - \alpha \nabla + \frac{\alpha(\alpha - 1)}{2!} \nabla^2 - \right. \\ &\quad \left. \frac{\alpha(\alpha - 1)(\alpha - 2)}{3!} \nabla^3 + \dots \right] f_n \quad (17)\end{aligned}$$

What does this tell us? It tells us that if we have the function in its tabular form, and also its first, second, and higher backward differences, we can compute the function to any order we need to. And where do we get the differences? From the table for $f(x)$, since by definition:

$$\nabla f_n = f_n - f_{n-1} \quad (18)$$

and higher-order differences are merely more of the same:

$$\nabla^2 f_n = \nabla f_n - \nabla f_{n-1}, \text{ etc.} \quad (19)$$

We could use Equation 17 directly, but it's more efficient to take one more step to arrive at the form of Equation 10 and expand the powers of α and collect terms. This evaluation is a rather tedious process but it needs to be done only once. If you have a symbolic math program such as Mathematica or Mathcad, you can let it do all the work. You can also let the computer crunch the table of the function to generate the difference tables to any order. The end result will be a table of coefficients a_n , b_n , c_n , and so on, suitable for use in an equation like Equation 10.

I didn't exactly invent this method. It came a little before my time, from none other than Charles Babbage. Remember Babbage's difference engine? Now you know what differences it was computing. It was designed specifically to generate and evaluate functions like $\sin x$.

UNEVEN TABLES

Well, that was something of a digression, but I thought you should see the unifying relationship between the use of table lookups and power series. Now that we've done that, let's get back to table lookups. Until now, we've considered only the case where the table consists of values on a uniform grid. We didn't need to store the value of x because it's implied by the position in the table. In the more general (and more difficult) case, we're given the data as a set of ordered pairs $\{x, f(x)\}$.

We can still do a table lookup and linear interpolation as before:

$$f(x) = f_n + m_n(x - x_n) \quad (20)$$

in which m_n is still given by Equation 3. None of this changes when we go to uneven tables. What changes is our ability to deduce the index n for a

given entry. We can no longer just compute n from the argument x , we must go looking for it. I guess that's why they call the process "lookup."

In the remainder of this column, we'll look at different ways to accomplish this task. As usual, you'll find that some subtleties creep in that require careful attention to detail to get a good algorithm. For the purpose of illustration, I'll continue to use our sine function example, but in real life, we have a much more efficient way to evaluate that function.

FINDING THE RIGHT ENTRIES

Linear interpolation is really only interpolation if the two tabular points bracket the desired one:

$$x_n \leq x < x_{n+1} \quad (21)$$

If this condition isn't met, we'd be doing extrapolation, not interpolation. So our assignment, should we choose to accept it, is to find the two adjacent entries for which the conditions in Equation 21 are met.

The obvious (but slow) solution is to begin at one end of the table and check every entry. The algorithm is shown in the following pseudocode:

```
i = 0;
while(x[i] > x)
    i++;
```

We must also look out for end conditions, so some safeguards are in order to make sure we don't run off the table. A working function, including these safeguards, is shown in Listing 1. Let's look at how they work. The first one, the test against entry zero, simply makes sure that the input value x is at least as large as the first table entry. If it's not, we return the value corresponding to that entry. An alternative would be to flag this situation as an error in usage, but such error messages are of little value in the embedded applications we're supposed to be concentrating on here. We don't want to get "Alarm 1202" when we're 100 feet above the moon, as Neil Armstrong and Buzz Aldrin did. It's far better to

LISTING 1

Linear search.

```
#define SIZE 90

struct table_entry
{
    double x, y;
};

table_entry table[SIZE+1];

double lookup(double x)
{
    if (x < table[0].x)
        return table[0].y;
    for(int i=1; i<= SIZE; i++)
    {
        if(x <= table[i].x)
        {
            double x0=table[i-1].x;
            double x1=table[i].x;
            double y0=table[i-1].y;
            double y1=table[i].y;
            double slope=(y1-y0)/(x1-x0);
            return y0+(x-x0)*slope;
        }
    }
    return table[SIZE].y;
}
```

return our best guess and keep going.

If we pass our first safeguard, we know that our input value x is at least as large as the first table entry. Our next step is to make sure that it's smaller than the next entry, which will be our x_{n+1} . We keep stepping in the table until that condition is satisfied. If we fall out of the loop before returning, we know that x was larger than the largest table entry. That's our other safeguard, and in this case, we simply return the function value corresponding to the largest entry.

The key to this search algorithm and the reason we don't need to test both limits is that we always start from the beginning, so we know x must be larger than the current x_n or we would not be at the current index. This means that we can avoid one of the range tests.

As I've written the algorithm, I've used a number of intermediate variables. If those offend your sensibilities, we can easily get rid of them. Listing 2

shows a shorter, more efficient but also more obtuse algorithm.

For a more general-purpose algorithm, we can also pass the table and its size through the calling list:

```
double lookup(double x, function_table &
    table, int size)
```

Among other advantages, this approach lets us use the same lookup function to handle multiple tables.

SPEEDING UP THE SEARCH

The worst problem with the algorithm I've shown is the time it takes to find the right place in the table. When we looked up an entry in our old trig books, we could use some intelligence to narrow down the search, but the algorithm as shown must always begin at the very first entry. For a large table with 1,000 entries, we can expect to try 500 values on average before we find the right interval. Those kinds of numbers don't seem to make the algorithm very efficient. Any performance gain we get by using only a linear interpolation as opposed to a polynomial solution is bound to be swamped by the time required to locate ourselves in the table. Is there anything we can do to speed up the process?

One interesting solution presents itself during repetitive searches for something like a dynamic simulation. If the variables aren't changing too

LISTING 2

A shorter version.

```
double lookup(double x)
{
    if (x < table[0].x)
        return table[0].y;
    for(int i=1; i<= SIZE; i++)
    {
        if(x <= table[i].x)
            return table[i-1].y+(x-table[i-1].x)*
                (table[i].y - table[i-1].y)/(table[i].x- table[i-1].x);
    }
    return table[SIZE].y;
}
```

Because of the nature of the binary search, it's a likely candidate for a recursive implementation.

fast, it's reasonable to suppose that the area of the table we'll be looking in will not be far from the place it was the last time we looked. This possibility suggests that we keep a static variable pointing to the last table entry we used, and search from there. This time, we must allow for searches in either direction. Listing 3 has the code. Here, I've reverted to using the intermediate values just to make the code less daunting to read. Feel free to remove them again once you understand what the code is doing.

This algorithm has the advantage that it never looks farther afield than its last index to find the new range. For something like a dynamic simulation,

LISTING 3

Search with memory.

```
double lookup(double x){
    static int i = 0;
    while((x > table[i+1].x) && (i <
        SIZE - 1))
        i++;
    while((x < table[i].x) && (i > 0))
        i--;
    double x0 = table[i].x;
    double x1 = table[i+1].x;
    double y0 = table[i].y;
    double y1 = table[i+1].y;
    double slope = (y1 - y0)/(x1 - x0);
    if(x < x0)
        return y0;
    if(x > x1)
        return y1;
    return y0 +(x - x0) * slope;
}
```

we can expect that the typical call will not require any search at all; the new value of x is still likely to fall in the same interval. At most, one adjustment high or low is all we should expect. The result is a dramatic improvement in efficiency.

Of course, all that improvement goes out the window if the routine is called into service for more than one table lookup. Since we have no particular reason to expect that the two x values will have any relationship to each other, we can expect the efficiency to go to pot for such an application. Fortunately, the fix is easy: Maintain a separate index for each table and for each lookup into the table and pass it through the calling list. One appealing approach might be to create a C++ class for the table instead of a simple array and include the static index as one of the member data items. In that way, we can be assured that the index will be undisturbed between successive references.

BINARY SEARCH

Any good computer science graduate will tell you that, of all the ways to search a table for a value, a linear search is the worst. A far better approach involves a binary search. Here, we begin by looking at the midpoint of the table and deciding which way the desired interval lies. In general, we can expect a search of order $\log n$, rather than n —a huge improvement for large tables. The binary search can work for any sized table but works best when the table length is a power of two. The tests get a bit tricky because we must watch for pitfalls like falling off of one end of the table or the other or getting caught in some kind of infinite loop. Listing 4 gives an algorithm that works.

Because of the nature of the binary search, it's a likely candidate for a recursive implementation. The idea is to split the table into two parts and decide which half our given value of x must fall into. Once we determine that, our algorithm calls itself recursively using that half. The process ends when we're down to a table with only two

LISTING 4

Binary search.

```
double lookup(double x){
    double x0, x1, y0, y1;
    int mid;
    int i0 = 0;
    int i1 = SIZE;
    while(i1-i0 > 1){
        mid = (i0+i1)/2;
        if(x > table[mid].x)
            i0 = mid;
        else
            i1 = mid;
    }
    x0 = table[i0].x;
    y0 = table[i0].y;
    if(x <= x0)
        return y0;
    x1 = table[i1].x;
    y1 = table[i1].y;
    if(x >= x1)
        return y1;
    double slope = (y1 - y0)/(x1 - x0);
    return y0 + (x - x0) * slope;
}
```

entries, straddling the root. The recursive solution is shown in Listing 5.

REFINING THE GUESS

A final intriguing possibility has to do with where we should begin looking for the solution points. Statistically, splitting the table in half should give us the minimum number of trials, but we're not really looking for any kind of average best guess. In each trial, we're given an input number, and we want to guess where to look for it in the table. If the number is, for example, near the lower end of the table's range, it stands to reason we should look near the beginning of the table. Even though the table points are not uniformly spaced, we can still expect to find small numbers near the beginning and large ones near the end.

I've used this idea to generate the first guess for the code in Listing 6. I like to think of it as a guided binary search in which we use the range at each level to decide where to split the table. Not surprisingly, this implementation blows the competition away on my sine-function test case, but that's

LISTING 5

Recursive version.

```
double lookup(double x, struct
    table_entry *table, int n){
    double x0, y0, x1, y1;
    assert(n > 1);
    if(n == 2){
        x0 = table[0].x;
        y0 = table[0].y;
        if(x <= x0)
            return y0;
        x1 = table[n-1].x;
        y1 = table[n-1].y;
        if(x >= x1)
            return y1;
        double slope = (y1 - y0)/(x1 -
            x0);
        return y0 + (x - x0) * slope;
    }
    int mid = n/2;
    if(x < table[mid].x)
        return(lookup(x, &table[0],
            mid+1));
    else
        return(lookup(x, &table[mid], n-
            mid));
}
```

not a very fair test because in this case, the table is evenly spaced. Therefore, the guided search usually hits the right index on the first try. But even with an uneven table, its performance will be considerably better than a pure binary search unless the organization of the table is truly pathological.

TIME IS OF THE ESSENCE

Despite our best efforts at improving the performance of the lookup into an unevenly spaced table, we will never be able to match the performance of the one for evenly spaced tables; that algorithm nails the bracketing points on the very first try, leaving little room for improvement. So what do you do when you need very high performance but have an unevenly spaced table? My advice is to turn it into an evenly spaced one.

Build yourself the most exotic, most accurate table lookup routine you can think of with the highest-order interpolation scheme you can find. Just don't use it in your real-time application.

LISTING 6

"Educated guess."

```
double lookup(double x, struct
    table_entry *table, int n){
    double x0, y0, x1, y1;
    int mid;
    assert(n > 1);
    x0 = table[0].x;
    x1 = table[n-1].x;
    if(n == 2){
        y0 = table[0].y;
        if(x <= x0)
            return y0;
        y1 = table[n-1].y;
        if(x >= x1)
            return y1;
        double slope = (y1 - y0)/(x1 -
            x0);
        return y0 + (x - x0) * slope;
    }
    if(n == 3)
        mid = 1;
    else{
        mid = (int)((double)(n-1)*(x-
            x0)/(x1-x0));
        mid = max(min(mid, n-2), 1);
    }
    if(x < table[mid].x)
        return(lookup(x, &table[0],
            mid+1));
    else
        return(lookup(x, &table[mid],
            n-mid));
}
```

Instead, use it to scan your data at periodic intervals, the smaller the better, and store the results in a new table that's evenly spaced. In that way, all your heavy-duty number crunching gets done offline once and once only. After the second table is built, you can park the fancy lookup routine and stick to the easy and fast one. **ESP**

Jack W. Crenshaw is a staff scientist at Invivo Research in Orlando, FL. He did much early work in the space program and has developed numerous analysis and real-time programs. He holds a PhD in physics from Auburn University. Crenshaw enjoys contact and can be reached via e-mail at 72325.1327@compuserve.com.